Constrained Local Model for Face Alignment, a Tutorial

Version 0.7

By Xiaoguang Yan,
xiaoguang.yan@gmail.com
March 24, 2011.

Forewords

Finding faces in an image is useful task. You can search online to see how face detection/alignment technology is used in shooting *Avatar*.

Finding faces in an image is not easy, even difficult is finding position and shape of eyes, nose and mouth, etc. Figure 0 gives an example:

Figure 0, face alignment example

Many methods are available for this purpose, some of the best known are Active Shape Model (ASM), Active Appearance Model (AAM) and other variants. Currently the de facto standard method is AAM. Up until 2010, when this tutorial is written, AAM and its variants are still being actively studied, and papers are published in top-tier computer vision conferences such as ICCV, ECCV, and CVPR.

A new method, called Constrained Local Models (CLM), has emerged since 2006 as a promising new method for face alignment.

However, the papers so far published on CLM are mostly technical, with many formulas, and require good math background as well as good knowledge of face alignment field. For a beginner, starting to understand CLM by reading these papers can be a daunting task. Implementing his/her own version of CLM, of course, will be even more difficult.

It is for this reason that this tutorial is written, to provide readers with a basic understanding of the rationale and intuition behind CLM, and get readers ready for the advanced methods found in ICCV or journals.

One more thing that is lacking, in the papers published so far, are details on implementation. To implement CLM, one would have to study the papers and try to fill in the details on ones' own. This tutorial fills the gap between published papers and ones' own implementation by providing detailed description of implementation of CLM, so that reader can have a concrete understanding of CLM implementation, as well as the algorithm itself. Sample source code written for Matlab and OpenCV are also provided, for study and reference.

In order to make the tutorial easy to ready, I add some comments and some of my own observations. These are provided to convey the intuitions, instead of definitions, of the concepts. In some cases, rigorous mathematic treatment is omitted; interested readers are referred to the references listed in the end for a formal treatment of the topic.

Xiaoguang Yan
Changchun University

March, 2011

Note:

Sample images taken from FG-NET talking face video, available at:
http://www-prima.inrialpes.fr/FGnet/data/01-TalkingFace/talking_face.html

Table of Contents

1. Introduction to Face Detection and Alignment

Given an image, how can someone, say you and I, find a face, and further, its eyes, nose, etc? Of course there are a lot of theories trying to explain this process. One of the explanations could be: first take a quick scan of the whole image to determine which part of the image looks like a face, and then focus on that part, examining more closely to find where the nose, eyes, and mouth are.

Since we are interested in designing a computer program that can do face detection and alignment as human does, we can design our program to mimic this two-step process. We can implement the first step – a quick scan – with a Viola-Jones face detector [1]. This detector can find search an image for faces, and give rectangles of where the faces are found. To implement the second step – to look more closely at the rectangle of image, and determine the boundary of nose, mouth – is more difficult, and this is where CLM can help.

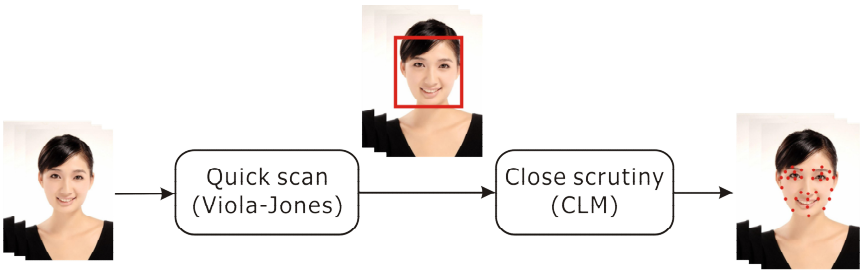

Figure 1, System architecture

We are in fact, detecting some specific points on face, such as eye and mouth corners, face contour points. Throughout this tutorial, we will call these points *feature points*. The job of CLM is to find these *feature points*, given a face image.

Fortunate for us, implementation of the first step, the quick scan, has been well studied, and you can find Viola-Jones detector implementations on Matlab, OpenCV, and even GPU. The purpose of this tutorial is to discuss implementation of the second step – finding eyes, nose and mouth – using CLM.

Now let's consider this: given a rectangular patch of image that contains face (and often, also its background), how does a real person find the mouth, nose, etc? Again there can be many explanations to this process. One of them could be: we have in our mind a rough idea of how a face might look like, for example (assuming the head is in its usual position), we know that a head is roughly elliptical in shape, with eyes on top left and right half, nose in the middle, and mouth at the bottom half; and in our mind, we also know what eyes and nose usually look like, so that we can find them in the patch of image.

There are two things to note in the process described above:

a. We already know what eyes/nose generally look like, otherwise we cannot find them even if they are in the patch of image. Or speak technically, we have a *model* of the look of eye/nose, and we use this model to search for them in a given patch of image.

b. We also know the arrangement of eyes/nose on a face, eyes on top, nose in the center, etc. We can think of this as a *constraint*. This constraint is good for us, because when asked to search for nose in a face image, we do not need to go the length of searching the whole image, only the center or a close neighborhood of center (*local region*).

In summary, we use *models* to search in local region around where the corresponding item might appear. And we also use knowledge of shape of a face to *constrain* the search. This is the rationale behind CLM. That is, we use *Local Models* to search for individual items, and use knowledge of the shape to *constrain* the search, hence the name *Constrained Local Models* (hopefully the inventor will forgive me if I misinterpret it).

Now that we understand the name of CLM, we also know that if we want to do CLM, we need to have two types of information. One is a model of what each individual item, such as nose, eyes, mouth (let us from now on call them *patches*) look like; we can think of it as a local *patch model*, i.e., for each item, we build a model of what each patch might look like, and use this model to search in the image. Another piece of information we need, is the where each patch can appear (of course for normal person, eyes are above nose, nose above mouth, etc.), we can think of this as a shape model. Later we can use this shape model constrain the search of individual patch.

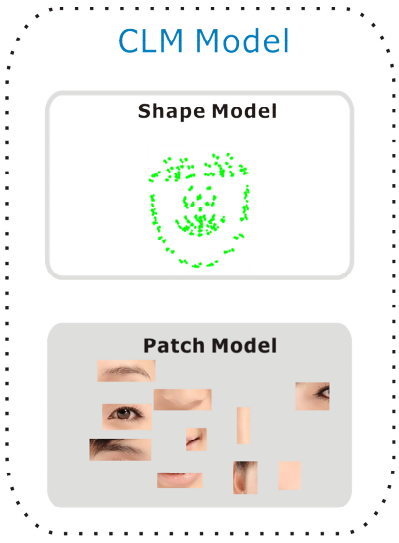

Figure 2, models in CLM.

Once we have these two pieces of information built as one CLM model, we are ready to find face and eyes and nose from a given picture. The idea is simple: since our shape model tells us the possible shape and arrangement of each patch, for each patch, we search where it most likely to appear. We also need to make sure we won't go beyond the position our shape model allows (or violate *shape constraints*, in technical terms).

So conceptually, the implementation of CLM consists of two steps:

a. Building a CLM model from a set of training images.
b. Use CLM to do search in new images.

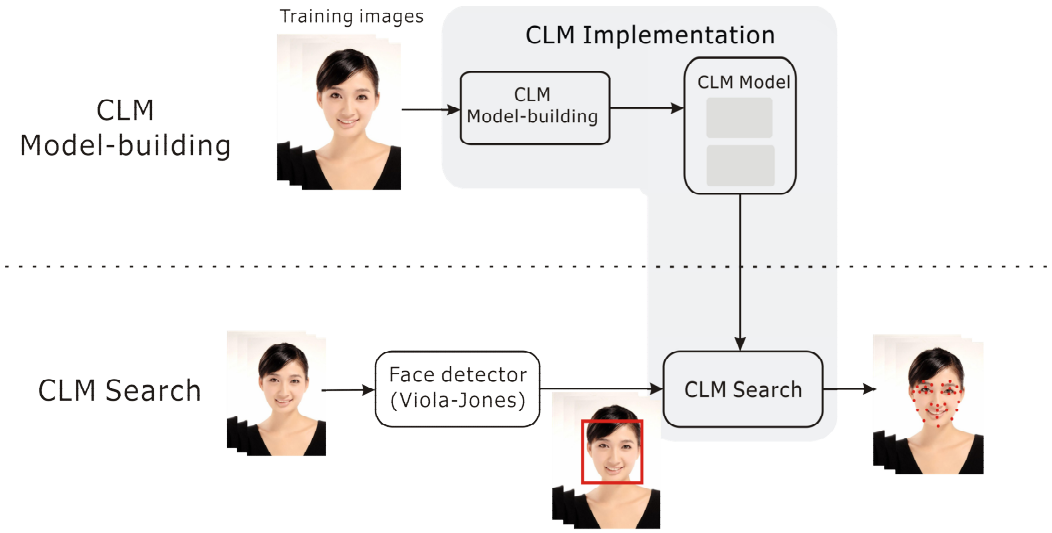

Figure 3, Conceptual diagram of CLM model and search

In this tutorial, we will call the first step model-building, and the second step the search process. Of course you will not think of them as totally independent processes: if you want to do good search, you need to design your model carefully, we will see this later in the tutorial.

That is it! Hopefully by now you have a rough idea of what CLM is, and how it is applied to search for eyes, nose and mouth.

Of course there are a lot more details to cover in this tutorial, so each of the following sections explains increasingly more details about CLM: in the next section, we discuss the architecture of CLM model-building and search in more detail; we focus on intuitions and rationale of CLM, so we intentionally keep it free of formulas. In section 3, we examine the formulas needed in CLM. Section 4 discuss implementation details. Section 5 concludes the tutorial by discussing some insights, CLM applications, and further improvements to CLM.

Section 1 summary:

- Finding faces from an image requires two steps: a quick scan, to find where might be a face; a close scrutiny, to find exactly where eyes, nose, mouth, etc. are.
- We can implement quick scan with Viola-Jones face detector, and close scrutiny with CLM.
- Before you can do close scrutiny with CLM, you need to have a face shape model, and a model of appearance of individual patch, i.e., a patch model.
- When you do close scrutiny with CLM, you search for patches around its most probable position, and use shape constraints to constrain your search.
- We will call the process of building a shape and patch model the model-building process, and CLM search the search process.

2. CLM Architecture

This section explains CLM in more detail. We examine how to build CLM shape and patch model, and how to do search with them. This chapter aims at giving you an intuition of CLM, so we intentionally keep it free of formulas. We will first explain CLM model-building, then the search process. A more detailed conceptual diagram of CLM implementation given below is a good outline of the discussion.

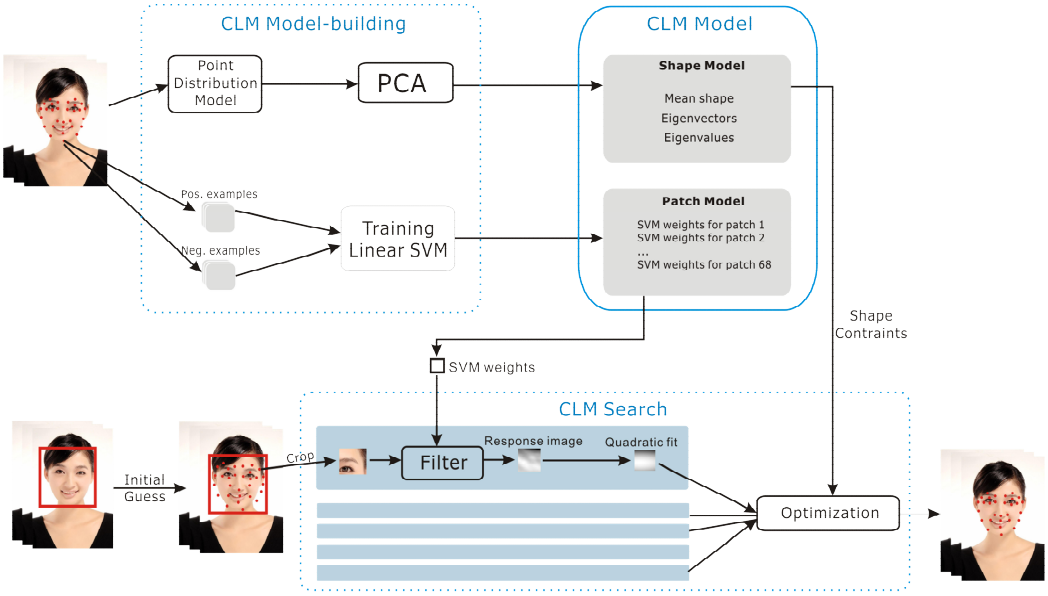

Figure 4, CLM model-building and search

We now examine more closely the model-building and search process.

2.1 Model-building:

Before we can use CLM to search an image for faces, we need to build a CLM model first. Remember building a CLM model is like learning what faces and each item look like, sometimes we call it *learning process*, in this tutorial we call it model building. Model-building can also be considered as training phase, in which we build a model from training images. As previously mentioned, a CLM model consists two parts, one part describing the shape variation of feature points, the *shape model*, and the other describing what each patch of image around the feature point might look like, the *patch model*.

Shape model describes how face shape can vary, and is often built with PCA. I assume you are familiar with the basics of PCA. You can find good tutorials on PCA online. Text box below discuss PCA specifics for building face shape models.

PCA for CLM shape model

Building shape model with PCA, given the training face shapes, usually consists of two steps.

First we calculate average of all shapes, to obtain a mean shape, which you can think of as common properties of all faces. We then subtract the mean shape from each shape; this gives us how each shape varies from the mean face. You can think of it as describing how much my own face differs from other faces, e.g. wide forehead, narrow eyes, long nose...

If you sit down and consider the number of variations human face could have, you might think its countless. But if we divide these variations into *types*, for example, wide face/narrow face, male/female face, Caucasian/Asian..., etc, we can say each of our faces can be a combination of these. For example we can say someone has a typical Asian male face, with narrow face and narrow eyes, etc. This gives much idea about what his face look like. From this intuition, can think of the *types of variation* as some basis, and every face is a combination of these faces, plus a mean face.

The next step of PCA is to find the basis of these variations, and calculate how much each basis represents all the possible variations. This can be done easily in Matlab, in two lines of code. (Yes, the *types of variations* are the eigenvectors, and the values of how much of the total variation they can account for, are their eigenvalue. We will talk about them more in later section.)

In summary, building a face shape model with PCA is the process of finding: the mean face shape, the basis of variations, and a value for each basis, representing how much variation each basis can account for.

If you are familiar with Active Shape Model or Appearance Model, you may know there is a preprocessing step, before doing PCA. That preprocessing is Procrustes analysis. Text box below explains why this preprocessing is needed. If you are familiar with Procrustes analysis, you can safely skip the text box below.

Preprocessing before doing PCA: Procrustes analysis

Think of this: of all the training images given, the faces could appear in different position in the image. For example, some may appear in the center, some in top left corner, or bottom (translation). Further, these faces could be of different size (scale), and some may be rotated (rotation). If we mark the feature points on these images, the feature points coordinates contains the translation, scale, and rotation information, as well as the shape variation.

Because CLM shape model is concerned with finding shape variation, not the translation, scale and rotation information, we need to find a way to remove those from the feature points coordinates. The common way to do it is Procrustes analysis.

You can think of Procrustes analysis as this: given a number of shapes, with different size, rotation and center position, we move, scale and rotate them to be roughly aligned.

After we remove the translation, scale and rotation, all that is left of the coordinates are the shape variations, and we are ready to do PCA on them.

Patch model describes how image around each feature point should look like. For example, we expect eyes to be an oval shape with a round pupil in the center, and each mouth corner should have its most likely appearance. Patch models can be built in several ways. Hopefully after finish reading this tutorial, you will be ready to devise your own approach as well.

In this tutorial, we build template model using linear Support-Vector Machine (SVM). In fact, for each feature point, we train a linear SVM to recognize the local patch around the feature point, and will later use it in the search process.

Again, I assume you know the basics of SVM. You can find tutorials on SVM online, but most of them contain many formulas. ☺ The text box below gives some intuition of SVM, but focus on using SVM for building patch model for CLM.

SVM for CLM patch model

Patch model describes how each patch around feature point should look like. We will use linear SVM to tell how each patch look like, or, say differently, tell us whether any given patch is the right patch or not.

The idea behind linear SVM is simple: suppose we are given a number of images (and suppose, for simplicity, these images are of the same size), and we are told whether each one is an image of nose or not. Is it possible to find a rule differential between nose images (positive examples) and non-nose images (negative examples)? We can think of this as finding a *boundary* (sometimes called *decision boundary*) between the positive and negative examples.

There could be many ways to draw such boundary. It is easy to observe: whether a patch contains nose or not (SVM output), has a lot to do with its image content, or intensities of pixels. So we can formulate SVM output as a linear function of its pixel intensities. So all we need to do now is to find the right weight for each pixel intensity value. This is SVM training. Building patch model with linear SVM is about finding the weights for each SVM, given the training images.

After we find the weights, we can use the weights to test new images and see if it contains nose or not.

Now that we are equipped with enough information, let us take on the challenge of finding face and nose, eyes, mouth, given an image.

2.2 Search process:

After building a CLM model, we can use it to find the position of nose, eyes and mouth in a rectangle image, we call it search process. Below lists steps in the search process (don't panic if you cannot understand, as we will explain these steps soon):

1. Make initial guess of feature point position.
2. For each feature point, use SVM to search in the local region of its current position, to obtain SVM response image.
3. Fit each response image with a quadratic function.
4. Find best feature point position by optimizing quadratic functions and shape constraints.
5. Repeat step 2-4 until converge.

We now explain these steps.

(Step 1-2) To do a search, we make a guess of initial position, possibly be initializing to the mean shape, then for each point, we use template matching method to search the local region around current position. The result is, for each feature point, we obtain a *response image* corresponding the local region searched. If we get high value in the response image, that means the match score is high, otherwise match score is low.

We then use these response images to determine the position of each feature point. You might be tempted to allow each point to jump to the position where highest response is found. But as mentioned before, this might lead to a weird shape; or spoken formally, *violate shape constraint*. So the job is to find the best position for each point from the response image obtained, taking into account the allowed shape variation, as described in the shape model. There are many ways to do this right.

(Step 3-4) Here is the crudest method: first find all positions with highest match score, check to see if it violates shape constraints, and if so, find a position that has a little lower match score, check the shape constraints again... until you find a position that does not violate shape constraint. This gives you the position that does not violate shape constraint, and still have a high match score. This method is crude, and NP hard, or mission impossible, in plain English.

> Example of an NP-Hard problem:
>
> Suppose, for example, if you have 5 feature points, and for each point you search within a 3x3 local region. This gives you 9 possible positions, in the worst case, you need to test your constraint... 9^5=50949 times until you find a solution, this may not seem much, but in real face alignment problem, we normally have 60+ feature points, and search within a local region of 16x16 or larger, so this worse case number becomes 256^{60} which, say you have a computer that starts calculating from the birth of the universe at 1 million calculations per second, and if you get really lucky, you find an answer next year, and still, this is only one iteration. We need to do several iterations to obtain the best result.

So is it possible to find the best position that gives highest match score, not violating shape constraints, at a speed of 30 frames per second, without exhaustively search all possible positions? Unfortunately, the answer is no, unless we simplify the problem.

Now think this: if the response image exhibits some type of feature, then maybe we can exploit such feature to speed up our search. For example, if the response image is in fact a 2-D Gaussian function, or a 2-D quadratic function, then we can exploit its properties when doing a search. Of course few, if any, of the response image is actually such a 2-D function. But if we can fit a function to the image, we can speed up the search. This will definitely degrade the search result, but we are hoping if we do the fit wisely, the result might be close to the real result.

So in this tutorial, we fit a 2-D quadratic function to each response image. We add these quadratics together (this might look weird, but I'll explain later), and call it the *response function*. We also write the shape constraints as a function of feature point positions, call it the *shape constraint function*. Adding these two functions together, we obtain a giant function, whose variables are position of each feature points. By optimizing this function (using off-the-shelf algorithms like quadratic programming), we can find the best position for each feature point.

(Step 5) Now that we have found the position of each feature point, it may seem we're done with the job. But there's another issue: if our SVM search is done on the whole image, maybe we can rest for now, but since our search is only done in a local region around current position, the result we obtain may only be a best one locally (local optimum). If we want to find the global best position, our safe bet would be to repeat step 2-4, until all points reaches a stable position. (The truth is, unfortunately, you are still not guaranteed to get a global optimum, due to the nature of face alignment problem. But this is as much as we can do for now.)

We have so far talked about building the shape model and template model; now let's jump into the math of CLM. You will love it, because it reveals the inner working of CLM model-building and search process.

Section 2 summary:

- CLM implementation consists of two processes: model-building and search process.
- A CLM model consists of a shape model and a patch model.
- To build shape model, we first use Procrustes analysis to remove rotation, scale and translation variation, then use PCA to find basis of variation and the amount of variation each basis represents (the eigenvectors and eigenvalues).
- To build patch model for each feature points, we train linear SVM with negative and positive samples, so that they can tell whether a given patch of image is the right one.
- CLM search processing goes like this: starting from an initial guess, we use linear SVM to search local region around its feature point, and obtain a response image. Next we fit a quadratic function to the response image. Then we can solve for new feature point positions by optimizing a function of quadratic functions and shape constraints. We repeat these steps until all points reach stable position.

3. CLM, the Math

As mentioned throughout this tutorial, to apply CLM on face alignment, we need to first build a CLM model, which contains the shape information and patch appearance information. Then you can use this information to search given image for faces.

This section discusses the math used in CLM model-building and search process. Subsection 3.1 go through the math required for building a CLM model, followed by 3.2 which discusses those used in CLM search process.

3.1 Building a CLM Model

Let us take a look at the problem of building a CLM: we are given a set of training images, these images of faces, and normally for each image, we are also provided with a separate file that contains the coordinate of feature points of faces in the image. This file is often manually labeled, meaning we have someone sitting at a computer, clicking on feature points.

> This is a non-trivial task! ☹
>
> If you have a thousand images, for each image you need to click 60 points, this will be a huge task. Perhaps you will think... well, can't we find some way to automate this? The answer is no... if we can automate this, then we won't need CLM☺.

We are now ready to jump into the math of CLM model-building.

3.1.1 Building Shape Model with PCA

Let's start with building shape model first. This includes two steps:

1. Remove scale, rotation and translation with Procrustes analysis, output aligned shapes.
2. Do PCA on aligned shapes.

Procrustes analysis:

I assume you are familiar with Procrustes analysis. You can find good tutorial online.

The idea of Procrustes analysis is simple: given two shapes, find rotation parameter θ, scale parameter a and translation (t_x, t_y), to minimize the mean squared error of between the first shape and second shape transformed . The optimal parameter values can be obtained analytically by taking partial derivatives of the mean-square error function.

Life is made easy by Matlab: there's a *procrustes* function in the toolbox, which takes two shapes and align them, you get the rotation, scale and translation parameters, together with the aligned shape. You can also find other implementation source code online.

Suppose we are given 100 training images with 100 face shapes. We can align all the shapes to the first shape, this removes scale, translation and rotation variation of the shapes, leaving us with only face shape variations. We can now do PCA on the aligned shapes.

Doing PCA on aligned shapes:

We use PCA to find basis of face variations. Our job is to find the basis of variations (eigenvectors) and a number indicating how much of the total variation this basis accounts for (eigenvectors).

The idea is this: suppose each face shape contains 60 feature points, and each feature point is given by its (x, y), this gives us 120-dimension vector for each shape. We then do this: for each 120-dimension vector, we first subtract its mean, then we put all 120-dimension vectors in a big matrix, and calculate the eigenvector and eigenvalue of its correlation matrix.

In math:

Suppose we are given M images, each image contains N feature points, and each feature point is specified by its (x, y) coordinate, we put all the feature point coordinates of an image into one vector:

$$x = \begin{bmatrix} x_1 & y_1 & x_2 & y_2 & \dots & x_N & y_N \end{bmatrix}^T$$

Of course for each image, we have an x, so we have M number of x, let's denote them by $x^{(i)}, i = 1, 2 \dots M$. We calculate the mean shape by:

$$\bar{x} = \frac{1}{M} \sum_{i=1}^{m} x^{(i)} \qquad (1)$$

then we subtract mean from every $x^{(i)}$, let's still denote them using $x^{(i)}$, just bear in mind that they now have zero-mean.

Now we stack all $x^{(i)}$ s' into one giant matrix:

$$X = \begin{bmatrix} x_1^{(1)} & y_1^{(1)} & x_2^{(1)} & y_2^{(1)} & \dots & x_N^{(1)} & y_N^{(1)} \\ x_1^{(2)} & y_1^{(2)} & x_2^{(2)} & y_2^{(2)} & \dots & x_N^{(2)} & y_N^{(2)} \\ & \dots & & & \dots & & \\ x_1^{(M)} & y_1^{(M)} & x_2^{(M)} & y_2^{(M)} & & x_N^{(M)} & y_N^{(M)} \end{bmatrix}$$

where each row of X is the vector $x^{(i)}$ from one shape.

We do PCA by calculating the eigenvector and eigenvalue of $X^T X$. We only keep eigenvectors corresponding to large eigenvalues.

> The common practice is to sort eigenvalues in descending order, and use a threshold value to cut off the last few eigenvalues, and drop their corresponding eigenvectors.

Now we have eigenvectors, and their corresponding eigenvalues. A common practice is to write eigenvectors as column vectors, say we have K columns of eigenvectors,

$p_j, j = 1, 2...K$. We can form a matrix of eigenvectors P, where each column of P is an eigenvector p_j. Let us use λ_j to denote the eigenvalue corresponding to the eigenvector p_j, $(j = 1, 2...K)$.

We now list two most frequently-used operations on eigenvectors below:

a. Shape decomposition:

Suppose we are given a new shape, let us still use $x = \begin{bmatrix} x_1 & y_1 & x_2 & y_2 & \dots & x_N & y_N \end{bmatrix}$ to denote it. We can write x as a linear combination of eigenvectors, plus the mean shape. Formally, we can write:

$$x = \bar{x} + PB$$

where $\bar{x}$ is the mean shape, calculated in (1), P is the eigenvector matrix, and $B = \begin{bmatrix} b_1 & b_2 & \dots & b_K \end{bmatrix}^T$ a column vector, which we will refer to as the *weight vector*.

If you are familiar with PCA, this should come as no surprise. You can think of $B = \begin{bmatrix} b_1 & b_2 & \dots & b_K \end{bmatrix}^T$ as a weight vector. Each element of B, $b_1 \quad b_2 \quad \dots \quad b_K$, is the weight of the corresponding eigenvector, and x is written as sum of eigenvector p_j multiplied by weight b_j, plus a mean shape.

We are often interested in finding the weight vector B, because it gives us some idea which eigenvector is dominant in x. A high value of b_j in B means the corresponding eigenvector is dominant in x. (Remember eigenvector is the basis of variation of all face shapes, so this tells us which type of variation is most frequently encountered.)

We can calculate B by:

$$B = P^T(x - \bar{x}) \qquad (2)$$

So given a shape, we can calculate the weight of each eigenvector. You can think of it as decomposing your shape into a linear combination of eigenvectors, and we calculate the weight of each eigenvector using (2).

b. Shape reconstruction:

This is easy: given a weight vector B, we sometimes wish to reconstruct the original shape x. This is done by:

$$x = \bar{x} + PB \qquad (3)$$

PCA and reconstruction error

Suppose we are given an x, we use decomposition to find B. If we use reconstruction, can we recover x? Not necessarily:

When we do PCA with $X^T X$, we get eigenvectors and eigenvalues. If we keep all the eigenvectors, and calculate B with them using (2), then we reconstruct x using (3), we get exactly x.

If we drop some eigenvectors, and calculate B then reconstruct x, we get a different value... The difference is caused by dropping some eigenvectors: if we drop some eigenvectors, and use the remaining subset of eigenvector to do the calculation, we are also dropping some elements of B, because each eigenvector produces one element of B. So the missing elements of B cannot contribute to the reconstruction. From this point, we can see the reconstruction error is exactly the portion of x that is projected on the dropped eigenvectors.

We will need this intuition later, when we formulate our objective function.

CLM shape model building is easy to understand, if you are familiar with PCA.

3.1.2 Building Patch Model

Let's continue with building a model to describe what each patch around the feature points look like.

As mentioned earlier, there are many ways you can describe what each patch looks like, for example, you can, for each patch, find a template that retains the common features of training patches, or you can use a decision tree to differentiate patch from background, or use SIFT, HOG, LBP, or other local descriptors to describe the patch and use classifiers to differentiate the right and not-so-right patches.

For this tutorial, we use training images to train linear SVM as patch *experts*. We call them patch experts because they are trained with positive and negative examples, and are good at telling if a patch is positive or not, like human experts do.

I assume you know the basics of SVM, if not, you can search online for tutorials on SVM.

About SVM:

SVMs are frequently used to tell whether something contains what you want to find or not. For example, if we want to use SVM to tell if an image contains a face or not, we first train a SVM with images containing face (often called *positive examples*), and image not containing face (often called *negative examples*). After SVM is trained, we can use it to tell if a new image contain face or not.

Idea: Suppose we have 1000 training images, each image contain 60 feature points. Then for each feature point, we have 1000 *positive training examples*. We can randomly sample elsewhere in the training image, to generate as many *negative examples* as we like. Then for each feature point, we train a linear SVM with the 1000 positive example, and the many negative examples we generated. This gives us 60 linear SVMs, each trained to tell whether a patch is *positive* or *negative*.

Math: Suppose for each feature point, we obtain m patch samples, some are *negative*, some are *positive* examples. All m patches have the same size, say total of n pixels. We concatenate each image data into one vector, say, by stacking the columns or rows. This way, we have m training vectors $\{x^{(1)}, x^{(2)}, ...x^{(m)}\}$, each training sample

$x^{(i)} = [x_1^{(i)}, x_2^{(i)}, ...x_n^{(i)}]^T, i = 1,2,...m$ is a column vector of n dimensions. Next we assign an output value for SVM: $y^{(i)} = \{-1,1\}, i = 1,2,...m$. If the training sample is a *positive* one, we assign $y^{(i)} = 1$, otherwise, we assign $y^{(i)} = -1$.

The most common notation of linear SVM is to express output as inner product of input data and support vectors:

$$y^{(i)} = \sum_{j=1}^{Ns} \alpha_j \left\langle x_j, x \right\rangle + b$$

where x_j is a subset of input training sample, called support vectors, and α_j the weight of the support vector, N_s being the number of support vectors, and b is a bias.

For CLM, this expression is not quite straight forward. Instead, we write SVM output as a linear combination of elements of input vector:

$$y^{(i)} = w^T \cdot x^{(i)} + \theta$$

where $w^T = \begin{bmatrix} w_1 & w_2 & ... & w_n \end{bmatrix}$ represent the weight for each element of input data, and θ is a constant, acting as a bias.

In CLM case, the set of training data are the patches extracted from the training images, and the output $y^{(i)}$ is set to 1 if the sample comes from its feature point position, and -1 if the data is randomly sampled elsewhere in the image. Using these data, we can train linear SVM, by finding w and θ.

From this formulation, we can see the job of training linear SVM, is just to search for the right w.

There is another twist in SVM: it is possible that no matter what value you chose for w, you cannot get the right output for the given input. In that case, we say the input is not linearly separable. So theoretically, there is no solution to the problem. But if we are willing to relax the requirement somewhat, say, we allow output to be not exactly -1, or 1, but some value between [-1, 1], then it is possible to find a solution of w.

> C parameter in SVM:
>
> In the formulas of such SVM, there is one parameter C that is particularly important: it reflects how much error you are will to relax the requirements to find w: if you set C very high (infinity, for instance), that means you want output to be exactly -1, or 1, but in case your data is not linearly separable, the search for w can go on forever; if you set C very low, you can always expect to find w, but the training error may be large.
>
> The C parameter is also in some sense related to *overfitting* and *underfitting*. In CLM case, If you set C very high, you are likely to find w that fits your training images well, but does not do well on other data (overfitting your training data). If you set C too low, your may end up with an SVM that has too high error even on training images (underfitting).
>
> Choosing a right value for C is an art that every SVM user should learn. ☺

By allowing output errors, we can find a solution to w and θ. Now that the output is not exactly -1 or 1, we can interpret the output as an indicator of how close output is -1, or 1.

There is one nice property of linear SVM: since the input to SVM, the x is, in fact, a vector concatenated from an image, and the output is an inner product of w (if we ignore bias θ) and x, if we instead collapse w into a 2-D image with the same dimension of x, we can think of w as an image filter that operates on the given x, outputs 1 if x contains what we want, or -1 otherwise. From this point of view, searching a local region with linear SVM, can be implemented as filtering the local region with filter w. This can speed up the search, as we will see later.

Now that we have built a CLM model by constructing shape model, and training linear SVM as patch model, we are ready to use the CLM model in the next section to search for faces.

3.2 Searching with CLM

Let us now deal with the math of searching with CLM. This is the most exciting part ☺.

Suppose we have already built a CLM model. Now we are given a new picture, how do we use the model to search for faces in the picture? The steps are already listed in the previous sections, and reiterated below:

1. First we use Viola-Jones face detector to find faces in the image. This gives us rectangles each containing one face.
2. Next we make an initial guess of the each feature point position. We can use mean shape for this purpose.
3. For each feature point, we crop a patch of image from its current position, and use linear SVM to find output at each point in the local region. This gives us a response image. We fit a quadratic function to the response image.
4. We find the best feature point positions by optimizing a function, created by combining the quadratic functions from step 3 and shape constraints from CLM shape model.
5. We move each feature point to its new position, and repeat step 3-5, until all feature points reach their best position, or we can do 3-5 for a fixed number of iterations, say 10, or 20 iterations.

Now to the math:

In step 1-2: there is not much math to go through in these two steps☺.

In step 3, for each feature point, we crop patch from its current position, use linear SVM to obtain response image, then fit a quadratic function:

a. Cropping a patch
   Assume we want to search within an 8x8 neighbor of current feature point position, and our SVM is trained on 10x10 patch. When we crop the patch from current position, we

should crop a (5+8+5)*(5+8+5) size image.

As mentioned in the previous section, if we use linear SVM, we can implement it as a filter working on the cropped image, and the filter output is the response image.

The response image is 8x8 in size, because it is also a 2-D image, we denote it with $R(x, y)$.

We then seek to fit a quadratic function to it.

b. Fitting a quadratic function

The idea is simple: find a quadratic curve that has minimum mean-square difference from response image.

In math:

Suppose $R(x, y)$ is the 2-D response image output from linear SVM. Suppose we have also identified the maximum value of $R(x, y)$, found at (x_0, y_0), and we wish to fit a quadratic curve at this position and minimize mean-square difference.

So our job is to find a curve in the form of:

$$r(x, y) = a(x - x_0)^2 + b(y - y_0)^2 + c$$

where a, b and c are the variables we need to find, to minimize the mean-squared error:

minimize $\varepsilon = \sum_{x,y} [R(x, y) - r(x, y)]^2$ (4)

We can, in fact, write it in matrix form, which is easier to handle. Let us concatenate $R(x, y)$ into a 1-D vector, as we did with image during SVM training,

$$R_L = [R_{11}, R_{12}, ..., R_{ww-1}, R_{ww}]^T$$

If we view ε as a function of a, b, and c, and introduce $\omega = \begin{bmatrix} a & b & c \end{bmatrix}^T$, then we can write (4) in matrix form:

$$\varepsilon = || A\omega - R_L ||_2^2 = \omega^T A^T A\omega - 2R_L^T A\omega + R_L^T R_L$$

And the problem is now:

minimize $\omega A^T A\omega - 2(A^T R_L)^T \omega$

Since the term $R_L^T R_L$ does not depend on ω, and can be dropped from the

minimization problem.

Since we also need to make sure the shape is a convex, not a saddle-like shape, we need to add these constraints: $a < 0$, $b < 0$.

So the final problem is:

minimize $\omega A^T A\omega - 2(A^T R_L)^T \omega$

subject to: $\begin{bmatrix} -\inf & -\inf & -\inf \end{bmatrix}^T < \omega < \begin{bmatrix} 0 & 0 & \inf \end{bmatrix}^T$

This problem can be readily solved with *quadprog* function in Matlab.

After the optimum ω is solved, $r(x, y)$ can be expanded as a function of x, y:

$$r(x, y) = X^T HX - 2F^T X + ax_0^2 + by_0^2 + c \quad (5)$$

where $X = \begin{bmatrix} x & y \end{bmatrix}^T$, $H = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}$, and $F = \begin{bmatrix} ax_0 & by_0 \end{bmatrix}^T$. We will need this form in the next step.

In step 4, we solve for the best feature point positions by optimizing a joint function of the quadratic functions from step 3, and shape constraints from CLM shape model.

The whole art is in formulating the function as one giant function that is complicated enough to reflect our objective, yet simple enough to be solved: if we write a function that reflects our object well, but is so complicated that there's no way we can find a solution, then we have no hope solving the problem; on the other hand, if we write a function that is very simple, and can be easily solved, but does not match our objective well, then it is just useless.

We aim at formulating our shape constraints and response image into a giant quadratic function, so we can solve for optimum feature point position using off-the-shelf quadratic programming methods. Let us now see how we can do this.

In fact, there are a number of ways you can choose to formulate your objective function. I will first introduce one that is based on [2], then another approach I adopted in my reference code.

Allow me introduce some notations first:

Assume we have $i = 1, 2...n$ feature points. For each feature point, we use SVM to obtain response and then fit a quadratic function $r_i(x_i, y_i)$ to the response image. We need to normalize these $r_i(x_i, y_i)$ to make them have the same max/min value, say within range [0, 1]. For brevity, we still use $r_i(x_i, y_i)$ to denote them. Just keep in mind they are now shifted and

scaled to range [0 1].

Now to the real business:

One approach, similar to [1]:

For CLM search, we are in fact searching for feature point position with high SVM response, while at the same time maintaining face shape; we can write the two goals into one function:

$$f(x) = \sum_{i=1}^{n} r_i(x_i, y_i) - \beta \sum_{j=1}^{k} \frac{b_j^2}{\lambda_j} \quad (6)$$

The first term is simply the quadratic response we fit. We wish to find (x_i, y_i) to maximize this term.

The second term is a shape constraint. Remember b_j is the weight for eigenvector j if the shape is decomposed, and λ_j is the eigenvalue corresponding to this eigenvector. You can think of the second term as: try finding the shape that is most close (in sense of *Mahalanobis distance*, in case you have noticed) to mean shape. So we add a minus sign to penalize a high value of this term. We also add a weight β to the second term, so we can adjust how much we penalize this term.

Together the function means: try find the shape that is most similar to mean shape, yet still give me a high response output SVM.

About β

If we set β to a high value, we are penalizing heavily on shape difference. So the problem solver we use will try to find a shape that is close to mean shape, possibly sacrificing (or even, completely ignoring) SVM response.

Otherwise, if we set β to a very low value, this is the same as telling your solver: 'I don't care so much about the shape constraints, just find me feature point positions with high SVM response.'

A sensible way to choose β is by calculating it from training data. But I like tweaking it a little, because it helps me gain an insight into the problem.

This function is, in fact, a quadratic form of feature point position, although not obvious at a first glance. So let us now rewrite it to an explicit form.

Let us first use $x = \begin{bmatrix} x_1 & y_1 & x_2 & y_2 & \dots & x_N & y_N \end{bmatrix}$ to denote the feature point vector we introduced before, and assume we have removed mean from x. We will write then first and second term as an explicit function of x, so that when we solve minimize the function, we get the best x, which is our new feature point position.

The first term:

Since we have fit quadratic function to each response image, we have (from (5), adding subscript i:

$$r_i(x_i, y_i) = \begin{bmatrix} x_i & y_i \end{bmatrix} H_i \begin{bmatrix} x_i \\ y_i \end{bmatrix} - 2F_i^T \begin{bmatrix} x_i \\ y_i \end{bmatrix} + a_i x_{0i}^2 + b_i y_{0i}^2 + c_i$$

that is, for the i-th feature point, the SVM output is fit to a quadratic function, H_i, F_i and $a_i x_{0i}^2 + b_i y_{0i}^2 + c_i$ together determines the shape of the quadratic function. If we introduce, and drop the $a_i x_{0i}^2 + b_i y_{0i}^2 + c_i$ term (it is not affected by (x_i, y_i), hence will not affect optimization result):

$$H = \begin{bmatrix} H_1 & 0 & \dots & 0 \\ 0 & H_2 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & H_n \end{bmatrix}, \quad F = \begin{bmatrix} F_1 \\ F_2 \\ \dots \\ F_n \end{bmatrix},$$

then the first term can be written as:

$$\sum_{i=1}^{n} r_i(x_i, y_i) \rightarrow x^T H x - 2F^T x$$

This is an explicit quadratic form of x.

The second term:

If we introduce $b_{nr} = \begin{bmatrix} b_1 / \sqrt{\lambda_1} & b_2 / \sqrt{\lambda_2} & \dots & b_k / \sqrt{\lambda_k} \end{bmatrix}$, we can in fact write the second term as:

$$-\beta \sum_{j=1}^{k} \frac{b_j^2}{\lambda_j} = -\beta b_{nr}^T b_{nr}$$

Interestingly, if we further introduce $P_{nr} = \begin{bmatrix} p_1 / \sqrt{\lambda_1} & p_2 / \sqrt{\lambda_2} & \dots & p_k / \sqrt{\lambda_k} \end{bmatrix}$, that is, P_{nr} is formed by dividing each column of eigenvector matrix P with square root of it

corresponding eigenvalue, we have:

$$b_{nr} = P_{nr}^T \cdot x$$

Then the second term can be written as:

$$-\beta \sum_{j=1}^{k} \frac{b_j^2}{\lambda_j} = -\beta b_{nr}^T b_{nr} = -\beta (P_{nr}^T \cdot x)^T (P_{nr}^T \cdot x)$$

Now the whole function is written as:

$$f(x) = \sum_{i=1}^{n} r_i(x_i, y_i) - \beta (P_{nr}^T \cdot x)^T (P_{nr}^T \cdot x) = x^T H x - F^T x - \beta x^T \cdot P_{nr} \cdot P_{nr}^T \cdot x \qquad (7)$$

which is indeed an explicit quadratic function of x.

(These many formulas are hard to swallow at one time, take your time to verify for yourself...☺)

Let us consider the shape constraints formulated in this function: it says if we project our shape over the eigenvectors of shape model, we want the weight (divided by eigenvalue corresponding to this eigenvector) to be as small as possible.

But here is a question... how about the reconstruction error (see *PCA and reconstruction error* text box in section 3.1.1)? Theoretically (and possibly in practice), it is possible to find a shape that has a small value on the second term in (7), but has a large reconstruction error. If that happens, you get a shape that is heavily deformed because of the reconstruction error, but still your optimizer seeks to deform it even more, because nothing stops it from doing it.

So I seek another approach...

My approach:

We seek to minimize the reconstruction error, while at the same time, keeping $-3\sqrt{\lambda_j} < b_j < 3\sqrt{\lambda_j}$, to maintain shape constraints.

In math: if we write reconstruction error as the different between the real x and reconstructed x:

$$\varepsilon_{rec} = x - PP^T x$$

we can formulate our objective function as:

$$\tilde{f} = \sum_{i=1}^{n} r_i(x_i, y_i) - \beta (x - PP^T x)^T (x - PP^T x)$$

subject to: $-3\sqrt{\lambda_j} < b_j < 3\sqrt{\lambda_j}$

Compare with (6), we can see that, instead of trying to minimize b_j^2/λ_j, we now explicitly state that $b_j^2/\lambda_j < 9$ as a condition that must be satisfied (the condition $-3\sqrt{\lambda_j} < b_j < 3\sqrt{\lambda_j}$). In the objective function, we now add a penalty term of reconstruction error. We now seek to maximize SVM response, minimize reconstruction error, and at the same time, keep b_j^2/λ_j in a reasonable range.

Again, if we introduce $P_{nr} = \left[p_1/\sqrt{\lambda_1} \quad p_2/\sqrt{\lambda_2} \quad \ldots \quad p_k/\sqrt{\lambda_k} \right]$ as we did before, we can write the constraint as:

$$-3 < P_{nr}^T \cdot x < 3$$

and we adopt $\sum_{i=1}^{n} r_i(x_i, y_i) \rightarrow x^T H x - 2F^T x$ as before, the final problem can be written as an explicit function of x:

maximize $\tilde{f} = x^T H x - F^T x - \beta(x - PP^T x)^T (x - PP^T x)$

subject to: $-3 < P_{nr}^T \cdot x < 3$

Again, this is a quadratic programming problem, and can be readily solved using Matlab *quadprog* function. (Due to implementation considerations, this is not the actual formulation we use. The actual formulation we use is given in section 4.)

Summary:

I cannot think of a way to summarize all these formulas into a few lines☺. I just hope you can see through the formulas what we intend to do to obtain the CLM model, and carry out the search. If you cannot understand some formulas, don't freak out. It happens to most of us. Just take your time... Anyway, here are a few lines you can note on:

- The problem of fitting a response image to a 2-D quadratic is solved by quadratic programming with constraints.
- We formulate shape constraints as a quadratic function, so that when combined with 2-D quadratic response, form a giant quadratic function. We find the optimum using quadratic programming, and obtain optimum feature point positions.

Hopefully by now you have a better understanding of CLM model-building and search. In the next section, we'll discuss some issues we need to consider during implementation, and analyze implementation details.

4. Implementing CLM

This section discusses CLM implementation details. Again, following the organization of previous sections, we first discuss CLM model-building, followed by CLM search process implementation.

4.1 CLM Model-Building Implementation

Let us see how to implement CLM model-building. As mentioned previously, CLM model consists of a shape model and a patch model. We will discuss building shape model first, then patch model.

Building shape model

Assume we have 1000 images, called training images. Each image comes with a file giving the coordinate of feature points from faces in these images. As mentioned before, these files are generated by someone sitting before computer screen clicking on feature points. Suppose we use 68 feature points each face. We will build shape model with these feature point coordinates.

We will first use Procrustes analysis to align these shapes, and then use PCA to find eigenvectors and eigenvalues. Here are the details:

First, we put coordinate of the 68 points into one vector, as described in section 3. Since each point is an (x, y) pair, we have a face vector of 136 numbers for each face shape. We use Procrustes analysis to align the vectors. Suppose we align them all with the first face vector. Doing Procrustes analysis in Matlab is easy: *procrustes* function can do the job. The inputs are two shapes, and output is the 2nd shape aligned with the 1st shape.

Now we have a number of new face vectors, each vector a new shape obtained by aligning face shapes with the 1st face shape, we can do PCA on these new shapes. Suppose the vectors are column vectors. We form a giant matrix by putting the first shape vector in the first column, 2nd shape vector in the 2nd column… We have a giant 136x1000 matrix. This matrix is the input to PCA.

Doing PCA with Matlab is easy: you can use *eig* function, or you find PCA implementation online. After doing PCA, you will get: mean shape vector, eigenvalues, and eigenvectors.

Your shape model is now ready.

Building patch model

This is a little more complicated, but nothing mysterious.

Our job is to extract patches from the 1000 training images, and use these patches to train SVM. Let's see how we can do it.

First, if we want to train SVM, we have to use fixed patch size, because you cannot train SVM with one set of data of 256 dimensions and later want to test it with data of 128 dimensions. So let's first fix patch size: suppose we use patch size of 16x16.

Considering each face has 68 points, we crop 68 patches from each face. But there is a complication: we have to take into account that these faces can be of different sizes, and orientations. A 16x16 patch on a small face image might cover half a face, while in the case when

a face image is large, a 16x16 patch can only cover a tiny area.

So we need to scale and rotate them into roughly the same size before can do cropping. Let's say we scale face to fit in the 300x400 pixels rectangle, nose in the center, in the usual orientation head is in. After we do this, we can safely crop patch from feature point position.

For each face, from it 68 feature point positions, we crop 68 patches of 16x16 pixels. Given 1000 training images, for each feature points, we have 1000 patches. These 1000 patches are positive training examples for SVM. We can randomly sample elsewhere in the image to generate negative training examples, but we choose to sample at positions near the feature point position, because later we want to train SVM to be able discriminate between patch from exactly the given feature point position, and those from nearby position. We can crop as many negative training examples as we like, the more the better SVM performance, at the price of slower training process.

Now that we have positive samples and negative examples, we can train SVM. If we use Spider SVM (available from: http://people.kyb.tuebingen.mpg.de/spider/), this is easy. For each feature point, we construct an SVM; so we have 68 SVMs. For each patch we cropped, we concatenate it into one vector, this is our x's. For each x, we assign an expected output from SVM: output y=1 if the patch is a positive example; or y=-1 if patch is a negative one.

With Spider SVM, we call *train*, passing the training data, and expected output, to train SVM. After SVM has been trained, we obtain weights by calling *get_w*. Because we use 16x16 pixel patches, we get 256 weights. We collapse the weights into a 16x16 patch for use in the search process.

The SVM weights are all the information we need for patch model, for now.

4.2 CLM Search Implementation

Now let's talk about CLM search. As mentioned in previous sections, we use CLM after Viola-Jones face detector. The output of V-J face detector is a rectangle containing a face. The task of CLM search is to find in this rectangle the position of each feature point.

Here are the steps to do it:

0. First we make an initial guess. We use mean shape as our initial guess. We scale, translate (and if necessary, rotate) mean shape to fit the rectangle, we keep the scale, translation (and rotation) factor for use later.

   From this initial guess, we take iterative approach to find the optimal feature point position.

1. For each feature point, we crop a patch from its current position. The size of the patch should be: (current scale factor)x(SVM patch size + search region size). For example: if in step 0, we scale mean shape by 0.5, we use SVM patch of 16x16 pixels as before, and we want to use SVM to search in the region of 4x4 around its current feature point position. Then in current image, at each feature point position we crop a patch of 0.5x(16+4) = 10 (+1 or -1 pixel, let us drop it for brevity) in size, and scale it to (16+4)x(16+4) in size.
2. For each feature point, we use the weight obtained from *get_w*, collapsed into 16x16

patch as a 2-D filter to filter the cropped image. This is done in Matlab with function *filter2*. The output is the response image. (In fact, we also need to add a bias to obtain the real response. But since we need to normalize the response later, we can drop the bias for now.) We normalize the filter output, keep them in range [0 1], and rescale it back to 4x4. (Since we are searching in a 4x4 area, our response image should be 4x4 in size.)

3. For each normalized response from 2, we fit a quadratic. This gives a, b and c for each response image.

(Steps 1-3 are good candidates for parallel speed-up.)

4. Now we are ready to use optimization to find the best position, and this is the tricky part. We will try to use (7) to find x.

maximize $\tilde{f} = x^T Hx - F^T x - \beta(x - PP^T x)^T (x - PP^T x)$ (7)

subject to: $-3 < P_{nr}^T \cdot x < 3$

The optimal x gives the best position, and is what we want to find. But there is a problem with this: when we fit quadratic in step 3, each response is fit to a quadratic in its local coordinate (say for each 4x4 response image, we fit a quadratic at x,y coordinate of [0 4]). But the shape constraint requires a global coordinate (coordinate of all feature points to their common origin). If we simply plug H here, the coordinate systems of the first two terms and the last terms are not different, and the result will be wrong. We need to reformulate (7) to resolve this.

We can, in fact, write (7) as:

$$\tilde{f} = x^T Hx - F^T x - \beta(\tilde{x} - PP^T \tilde{x})^T (\tilde{x} - PP^T \tilde{x}) \quad (8)$$

where $\tilde{x} = x + base$

That is, we add $base$ to x in the last term. The rationale is: if we want to keep the first term, then the second term will have to be modified, because the quadratic function given by the first two terms is in relative to its current feature point position, the second term is a function of feature point position difference from mean shape. So we add a $base$ term to the last term, to adjust for this difference.

But what is $base$? If we align current shape with mean shape using Procrustes analysis to obtain $aligned\_x$, then $base$ is simply

$$base = aligned\_x - mean$$

Now the justification, stated simply: if we add $aligned\_x$ to x we have new

feature point position (remember x is the deviation from current position, and $aligned\_x$ is current feature point position so $(aligned\_x + x)$ is the new position of each feature point); if we then subtract $mean$ from it, we get the shape difference, which is readily the input to the last term.

In summary, in step 4, we do:

a. Align current shape with mean shape.
b. Do optimization with (8).
c. Align back to get new feature point position.

All right, that is all the nitty-gritty of CLM. Hopefully by now you have a solid understanding of CLM and its implementation details, and is ready to analyze the sample code, or even write your own implementations. Good luck! ☺

5. Discussions

This section discusses some insights and possible directions for improvements. Some of these may be incorrect, and are provided just for your information.

5.1 Insights to CLM

Some of the points below come from my personal observation, and may not be totally correct:

1. Fitting quadratic function to response image is just one of the many methods to simplify the optimization problem. You can as well use other methods toward this goal. For example, you can think of using Gaussian function, or mixture of Gaussians to fit response image. You can use KDE as well [4].

2. When we do CLM search, the PCA shape reconstruction error and the projection b_j / λ_j together form the feasible space of our shape model: when we build a shape model with PCA, we find the eigenvectors and eigenvalues. Later in CLM search, when applying shape constraints, we need to determine how far off the data can go from the training model. For example, we need to set a threshold to shape reconstruction error, and b_j / λ_j (you can think of b_j / λ_j as describing how far away the shape is from your mean shape in the direction of the j-th eigenvector). So you can imagine creating a space, consisting of all the allowed value of b_j / λ_j and reconstruction error. It is within this space that our target shape is allowed to vary. So this space forms the feasible set of the optimization problem.

3. Parameter C that you can choose during linear SVM training is closely related to overfitting and underfitting. Below gives input images, C value, and corresponding linear SVM weights.

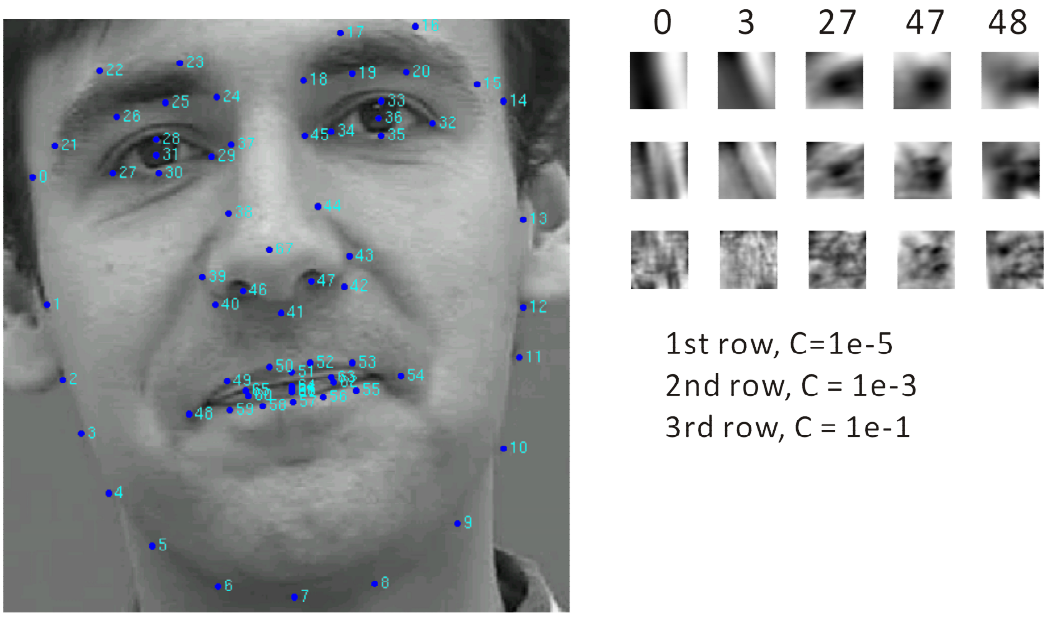

Figure 5, C values and SVM weights. Left image gives face image and numbering of feature points. Right images are SVM weights of feature point 0, 3, 27, 47, and 48, trained with different C. First row are trained with C=1E-5, second and third rows are trained with C=1E-3 and C=1E-1, respectively.

From this image, you can see that when C is low, that is, we allow SVM to make large training errors, the weights looks like a template. So later when you use SVM to do search, it is like doing template matching. Of course you may argue doing template matching is not enough, and SVM is *underfitting* your data.

When C is high, we have weights looking less and less like template and instead taking up more high frequency components. The high frequency components can sometimes be noise, taken by SVM as discrimination criteria. In this case, you might experience *overfitting*.

5.2 Possible Improvements

Below are the points I can think of, some of them may not be good directions.

1. Alternative simplification methods: as mentioned in the 5.1, you can use other methods to simplify optimization. [4] gives several methods. You can certainly devise your own.
2. Better local classifier: there is a limit to the discriminative power of linear SVM. So if you want to do better in local scale, you can replace SVM with other classifiers, such as Adaboost or decision tree. You can also couple them with better local feature descriptors, such as SIFT, HOG, LBP... Theoretically these should give you better local performance, possibly at the price of higher computational cost. Globally, you can always keep the global shape constraints, and use optimization to solve for optimum feature point position.

3. Parallel processing: the task of local patch cropping, SVM searching and quadratic fitting are carried out on each feature point individually, so this phase is ideal for parallel processing. You can use OpenMP to speed this up, as is done in the OpenCV reference code published on my website.
4. PPCA vs. PCA: in model-building phase, we use PCA. PPCA may improve the performance, because it takes into account the possible error introduced in the labeling process.

References

[1] P.Viola andM. Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR*, volume 1, pages 511–518, December 2001. 6

[2] D. Cristinacce and T. F. Cootes. Feature Detection andTracking with Constrained Local Models. In *EMCV*, pages 929–938, 2004.

[3] Y. Wang, S. Lucey, and J. Cohn. Enforcing Convexity for Improved Alignment with Constrained LocalModels. In *CVPR*, 2008.

[4] J. Saragih, S. Lucey, and J. F. Cohn. Subspace constrained mean shifts. In *ICCV*, 2009

Remark:

Ref. [2] is the original paper on CLM. Ref. [3] and [4] are modified versions of CLM.